Efficient lambda encodings for Mendler-style coinductive types in Cedille

Chris Jenkins, Aaron Stump, Larry Diehl

August 30, 2020

University of Iowa, Dept. of Computer Science

Introduction

Coinductive types and their lambda encodings

- Coinductive (coalgebraic) types classify (possibily) infinite objects. They come with
  - **Destructors** for making finite observations
  - **Generators** for their production (*corecursion schemes*)
  - And we often desire **productivity** for these corecursion schemes (i.e., all finite observations are well-defined)
- **Lambda encodings** are an identification of the codatatype with a particular corecursion scheme
  - Codata are lambda expressions
  - Types defined impredicatively ("impredicative encodings")
- Productivity for a given scheme comes **for free** if the corresponding encoding is definable in a total type theory

Why not adopt encodings where this matters? (total FP, ITPs)

Lambda encodings: some difficulties

Historically, lambda encodings for inductive types faced difficulties (**efficiency** and **logical expressivity**). Similarly for coinductive types.

Efficiency of corecursion for streams (in # of observations)

- **Church**: constructors incur **linear overhead**

  Constructors trigger re-generation of codata
- **Parigot**: **flat linear overhead**

  Additional case distinction for each observation

Overhead can be much worse for other coinductive types...

Datatypes as primitives

Inefficiency, and the lack of (co)induction in CC (**logical expressivity**), motivates the development of the *calculus of (co)inductive constructions* (CIC), where primitive (co)inductive datatypes are added to the theory.

Primitive datatypes with induction swell the TCB (positivity checking, possibly termination checking). Support for more expressive (co)recursion schemes may require further changes to the meta-theory.

Lambda encodings: some remedies

CDLE

The *calculus of dependent lambda eliminations* (CDLE) is a formally small extension of Curry-style CC that addresses the foregoing issues for lambda encodings directly

- induction is derivable *generically* for many encodings
- it possible to define *efficient* encodings for inductive types

Cedille is a higher-level language with special syntax for inductive types elaborated to lambda terms in CDLE.

What about coinductive types?

Lambda encodings: some remedies

CDLE

The *calculus of dependent lambda eliminations* (CDLE) is a formally small extension of Curry-style CC that addresses the foregoing issues for lambda encodings directly

- induction is derivable *generically* for many encodings
- it possible to define *efficient* encodings for inductive types

Cedille is a higher-level language with special syntax for inductive types elaborated to lambda terms in CDLE.

What about coinductive types?

This talk

A Mendler-style encoding for codata in CDLE that is

- **Generic:** works for any positive datatype signature
- **Efficient:** no penalty for constructors
- **Expressive:** supports a combined *course-of-values coiteration* and *primitive corecursion* scheme

Missing: true coinduction (bisimilarity $\Rightarrow$ equality), with a counter-result wrt CDLE's primitive equality type. Indexed corecursion for reasoning is supported (see the paper).

Focus of the talk

Focus: How we guarantee efficiency for *just* the primitive corecursion scheme.

- How the Mendler-style helps
- Monotone fixed point types (Matthes, 1998)
- Proof-irrelevant type inclusions (CDLE)

Mendler-style schemes for corecursion

What are “Mendler-style” corecursion schemes

For every “classic” structured corecursion scheme, there is an equivalent Mendler-style scheme

Advantages of the Mendler-style

- More idiomatic for FPers

  Explicit corecursive calls, like general corecursion

- Avoids intermediate structures in more complex schemes

  No build up / tear down assists in efficiency gain

Coiteration (streams)

Classic

$$\frac{h : S \to A \quad t : S \to S}{\texttt{coit}\ h\ t : S \to \texttt{Stream}\ A}$$

$$\texttt{head}\ (\texttt{coit}\ h\ t\ s) \rightsquigarrow h\ s$$
$$\texttt{tail}\ (\texttt{coit}\ h\ t\ s) \rightsquigarrow \texttt{coit}\ h\ t\ (t\ s)$$

Conceptually: $t : S \to S$ is a "state transition"

Mendler

$$\frac{h : S \to A \quad t : \forall R.(S \to R) \to S \to R}{\texttt{mcoit}\ h\ t : S \to \texttt{Stream}\ A}$$

$$\texttt{head}\ (\texttt{mcoit}\ h\ t\ s) \rightsquigarrow h\ t$$
$$\texttt{tail}\ (\texttt{mcoit}\ h\ t\ s) \rightsquigarrow \underbrace{t\ (\texttt{mcoit}\ h\ t)}_{R := \texttt{Stream}\ A}\ s$$

Conceptually: $t : \forall R.(S \to R) \to S \to R$ is a "generator transformer"

Primitive corecursion (streams)

"Short-circuit" generation by returning a pre-made stream

Classic

$$\frac{h : S \to A \quad t : S \to \texttt{Stream}\ A + S}{\texttt{corec}\ h\ t : S \to \texttt{Stream}\ A}$$

$$\begin{aligned}
&\texttt{head}\ (\texttt{corec}\ h\ t\ s) \rightsquigarrow h\ s \\
&\texttt{tail}\ (\texttt{corec}\ h\ t\ s) \rightsquigarrow \texttt{case}\ (t\ s)\ \texttt{of}\ \texttt{in}_1(x) \Rightarrow x \mid \texttt{in}_2(y) \Rightarrow \texttt{corec}\ h\ t\ y
\end{aligned}$$

Observations require **additional case distinction**

Mendler

$$\frac{h : S \to A \quad t : \forall R.(\texttt{Stream}\ A \to R) \to (S \to R) \to S \to R}{\texttt{mcorec}\ h\ t : S \to \texttt{Stream}\ A}$$

$$\begin{aligned}
&\texttt{head}\ (\texttt{mcorec}\ h\ t\ s) \rightsquigarrow h\ s \\
&\texttt{tail}\ (\texttt{mcorec}\ h\ t\ s) \rightsquigarrow \underbrace{t\ (\lambda\, x.\, x)}_{R := \texttt{Stream}\ A}\ (\texttt{mcorec}\ h\ t)\ s
\end{aligned}$$

Additional β-redex can be removed in CDLE

Monotone recursive types and proof irrelevant type inclusions in CDLE

Review: recursive types

The impredicative encoding obtained from a direct reading of `mcorec` requires recursive types.

$$\frac{h : S \to A \quad t : \forall R.(\texttt{Stream}\ A \to R) \to (S \to R) \to S \to R}{\texttt{mcorec}\ h\ t : S \to \texttt{Stream}\ A}$$

Recursive types

For a type scheme F, a recursive type μF is one such that:

- exists **roll** : $F\ (\mu F) \to \mu F$
- and exists **unroll** : $\mu F \to F\ (\mu F)$
- such that **unroll** (**roll** t) reduces to t (β-law)

Positive recursive types

Unrestricted recursive types lead to **non-termination**, with the culprits schemes F with negative occurrences of their type argument.

Can use syntactic notion of positivity to recover termination, or…

Monotone fixedpoint types

- f monotone iff $\forall x, y. x \leq y \implies f(x) \leq f(y)$
- Generalize **monotonicity** to type theory
  Need to interpret $\leq$, $\implies$

Representing monotonicity

Can interpret monotonicity in System **F**, but for recursive types this leads us back to **Church encodings** (β-law not satisfied).

CDLE twist: proof-irrelevant type inclusions

Preorder	$s \leq t$	$\forall x, y. x \leq y \implies f(x) \leq f(y)$
System F	$S \to T$	$\forall X, Y.(X \to Y) \to F\ X \to F\ Y$
CDLE	$\texttt{Cast}\ S\ T$	$\forall X, Y.\texttt{Cast}\ X\ Y \Rightarrow \texttt{Cast}\ (F\ X)\ (F\ Y)$

Erasure and irrelevance in CDLE

- CDLE is *Curry-style*: type annotations are external to the "real" term language (untyped lambda calculus) and are all erased
- Definitional equality is "up to erasure"

  $|\Lambda X.\, \lambda x{:}X.\, x| = \lambda x.\, x = |\lambda x{:}T.\, x|$
- Term arguments to functions can also be irrelevant
  - $t : T' \Rightarrow T$ means t is a function taking a T' argument, but **the T it returns does not depend on the choice of this argument**
  - for $t' : T'$, application $t \text{ -}t' : T$, and $|t \text{ -}t'| = |t|$

Proof irrelevant type inclusions

Type inclusions `Cast` S T are a derived notion in CDLE.

For brevity they are presented axiomatically (full defs. in appendix).

`Cast` S T **(elimination, erasure)**

$$\frac{c : \texttt{Cast}\ S\ T}{\texttt{elimCast}\ \text{-}c : S \to T} \quad |\texttt{elimCast}\ \text{-}c| = \lambda\, x.\, x$$

Takeaway: type inclusions are computationally irrelevant!

Monotonicity

$$\texttt{Mono}\ F =_{\mathbf{df}} \forall X, Y. \texttt{Cast}\ X\ Y \Rightarrow \texttt{Cast}\ (F\ X)\ (F\ Y)$$

We can derive

$\texttt{Mono}\ F$ **(elimination, erasure)**

$$\frac{m : \texttt{Mono}\ F \quad c : \texttt{Cast}\ S\ T}{\texttt{elimMono}\ \text{-}m\ \text{-}c : F\ S \to F\ T} \quad |\texttt{elimMono}\ \text{-}m\ \text{-}c| = \lambda\, x.\, x$$

Takeaway: monotonicity witnesses are computationally irrelevant!

Recursive types

`Rec` **(constructor, destructor, erasure)**

$$\frac{m : \texttt{Mono}\ F}{\texttt{roll}\ \text{-}m : F\ (\texttt{Rec}\ F) \to \texttt{Rec}\ F} \qquad |\texttt{roll}\ \text{-}m| = \lambda\, x.\, x$$

$$\frac{m : \texttt{Mono}\ F}{\texttt{unroll}\ \text{-}m : \texttt{Rec}\ F \to F\ (\texttt{Rec}\ F)} \quad |\texttt{unroll}\ \text{-}m| = \lambda\, x.\, x$$

- `roll` and `unroll` are part of a two-way type inclusion between $F\ (\texttt{Rec}\ F)$ and $\texttt{Rec}\ F$
- β-law easily satisfied!

The generic encoding

Generic variant Mendler encoding

$F : \star \to \star$ and $monoF$: $\texttt{Mono}\ F$ are parameters to the development

For simplicity, the below encoding has support for CoV coiteration *removed* (see paper for full definition).

Generic codatatype `Nu`

$$
\begin{aligned}
\texttt{CoAlg}\ S\ C &=_{\mathbf{df}} \forall R.\,(\texttt{Cast}\ C\ R) \Rightarrow (S \to R) \to S \to F\ R \\
\texttt{Nu} &=_{\mathbf{df}} \texttt{Rec}\ \underbrace{\lambda C.\,\exists X.\,X \times \texttt{CoAlg}\ X\ C}_{\text{positive in } C}
\end{aligned}
$$

- S is the "state space", C is the fixedpoint parameter
- Existential encoding is standard for codata
  (Church: $\exists X.\,X \times (X \to F\ X)$)

Efficient destructor

The type scheme in the definition of `Nu` is positive, so we can use `roll` and `unroll` to define the **generator** and **destructor** satisfying:

Generator and destructor

$$\frac{c : \texttt{CoAlg}\ S\ \texttt{Nu}}{\texttt{unfoldM}\ c : S \to \texttt{Nu}}$$

$$\texttt{outM}\ (\texttt{unfoldM}\ c\ s) \rightsquigarrow \underbrace{c\ \overbrace{\texttt{-castRefl}}^{\text{erased!}}}_{R:=\texttt{Nu}}\ (\texttt{unfoldM}\ c)\ s$$

No overhead for primitive corecursion!

Efficient codata constructor

$$\begin{aligned}\texttt{inM} \quad &: \quad F\ \texttt{Nu} \to \texttt{Nu} \\ &=_{\mathbf{df}} \texttt{unfoldM}\ (F\ \texttt{Nu})\ \Lambda R.\,\Lambda c.\,\lambda g.\ \overbrace{\texttt{elimMono}\ \text{-}\mathit{monoF}\ \text{-}c}^{F\ \texttt{Nu} \to F\ R}\end{aligned}$$

Ordinarily, use $g : F\ \texttt{Nu} \to R$ to corecursively re-generate. **But**

- $c : \texttt{Cast}\ \texttt{Nu}\ R$ and $\mathit{monoF} : \texttt{Mono}\ F$
- so $|\texttt{elimMono}\ \text{-}\mathit{monoF}\ \text{-}c| = \lambda\, x.\, x$

Consequently, $\texttt{outM}\ (\texttt{inM}\ t) \rightsquigarrow t$ **for all** t

- No re-generation of codata (Church)
- No traversal over F with `fmap` (Church and Parigot)

$\Rightarrow$ no penalty for constructors!

Conclusion

Summary and proviso

- Generic derivation of codata in CDLE supporting an expressive corecursion scheme efficiently.
- Required a different approach than used for efficient inductive types – *monotone recursive types á la Matthes*
- Termination guarantee in CDLE is somewhat subtle
  - Holds for closed terms which can be assigned a function type This includes all (closed) (co)data. Computing with open terms is in general unsafe
  - The features used to guarantee efficiency contribute to the proviso

Future work

- Ongoing work on an encoding with coinduction “up to” an equivalence relation derived from binary parametricty
  Rather than CDLE’s primitive equality type (used in the counter-example)
- Design of a nice surface-language syntax for codata (copattern matching, coercive subtyping using type inclusions)
  Mendler-style formulation helps bridge the gap between syntax and semantics: productivity checking reduced to type checking while maintaining a familiar style of explicit corecursive calls

Thanks!